

SISTEMA GUÍA PARA ESTUDIANTES DE PRIMER INGRESO

Documentación Fase#3 Proyecto-Iniciatec

Erika Cerdas Mejías - 2022138199

Leonardo Céspedes Tenorio - 2022080602

Kevin Chang Chang - 2022039050

Frankmin Feng Zhong - 2022089248

Entrega:
17 de Junio del 2024

Índice:

Diagramas oficiales de los patrones utilizados en la arquitectura.....	3
Patrón estructural utilizado.....	5
Elementos básicos y fundamentales de la arquitectura para los patrones Visitor y Observer implementados.....	9
Diagrama de clases (UML) actualizado.....	14
Cuadro de análisis de resultados de funcionalidades solicitadas.....	15
Cuadro de lecciones aprendidas individualmente.....	15
Repositorio de Git.....	18

Documentación de implementación de la fase 3 - Iniciatec

Diagramas oficiales de los patrones utilizados en la arquitectura

Diagrama de Patrón Visitor:

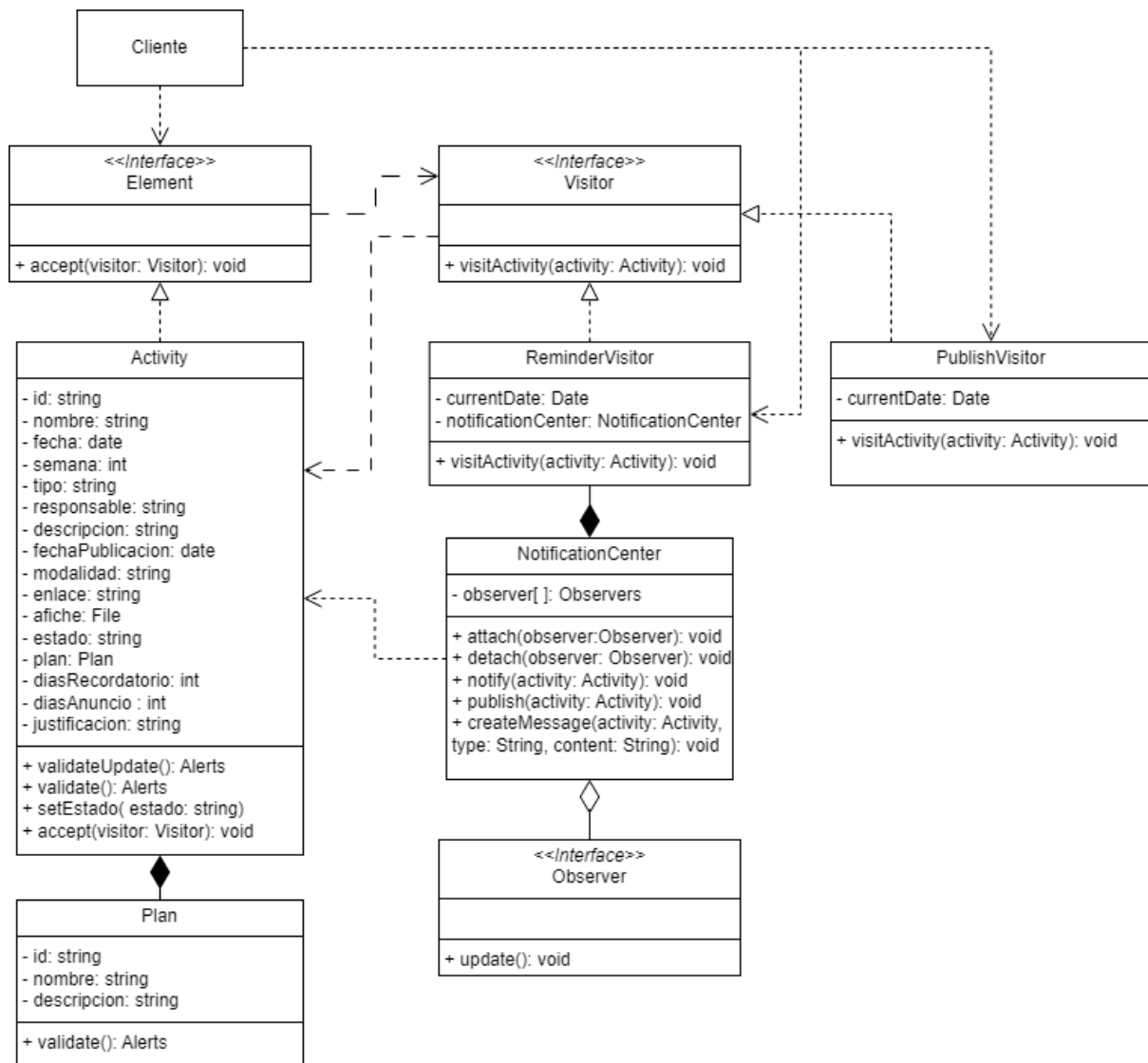


Figura 1. Diagrama de patrón visitor

Diagrama de Patrón Observer:

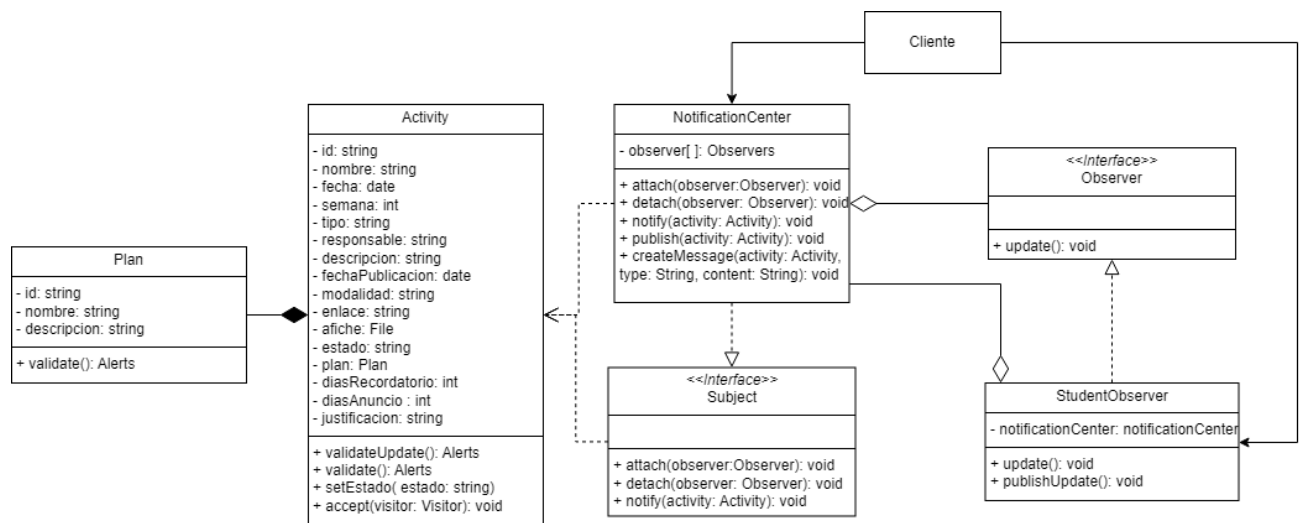


Figura 2. Diagrama de patrón observer

Patrón estructural:

Diagrama de Patrón Decorator:

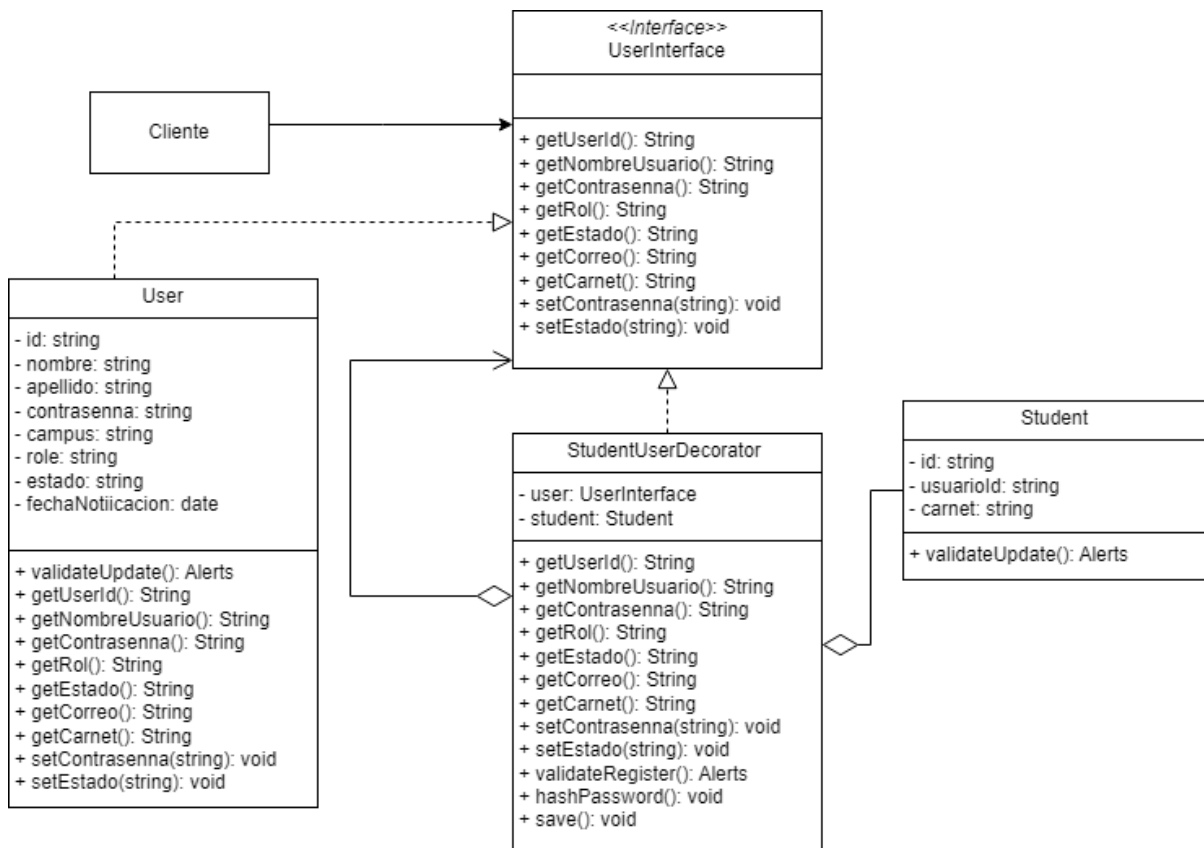


Figura 3. Diagrama de patrón decorator

Patrón estructural utilizado

Patrón estructural a utilizar:

Patrón Decorator

Justificación:

El hecho de usar el patrón decorador permite agregar funcionalidades adicionales a un objeto, sin alterar el código del objeto. En este caso, se crea un objeto llamado StudentUserDecorator que se encarga de crear una cuenta de usuario de cada estudiante registrado. Además, el patrón decorador facilita mantener separada la responsabilidad de transformación de estudiante a usuario. De esta manera, la clase User y Student conservan su responsabilidad principal, mientras que StudentUserDecorator se encarga de poder convertir al student en un user. Al usar el objeto StudentUserDecorator, se puede crear una instancia de User decorada con la funcionalidad de un Student. Incluso, el decorador puede ser reutilizado para agregar la misma funcionalidad a diferentes instancias de User,

Cambios:

- Se implementa la interfaz de UserInterface
- Se crea la clase StudentUserDecorator que implementa el UserInterface.
- Se actualiza en el controller de Students, la forma en la que se crean los usuarios de los estudiantes.

Evidencia de cambios:

En la Figura 3, se puede apreciar el diagrama de UML con la incorporación del patrón decorador. A continuación, se mostrará fotos de código con la incorporación de los cambios realizados.

```

1  class StudentUserDecorator implements UserInterface{
2      private $student;
3      private $user;
4
5      public function __construct($args=[]){
6          $this->student = new Student($args);
7          $this->user = new User($args);
8
9          $this->user->contrasenna = $this->student->carnet;
10     }
11
12     public function getUserId() {
13         return $this->user->id;
14     }
15
16     public function getNombreUsuario() {
17         return $this->user->correo;
18     }
19
20     public function getContrasenna() {
21         return $this->user->contrasenna;
22     }
23
24     public function setContrasenna($contrasenna) {
25         $this->user->contrasenna = $contrasenna;
26     }
27
28     public function getRol() {
29         $rol = Role::where('id', $this->user->roleId);
30         return $rol->nombre;
31     }
32
33     public function getEstado() {
34         $estado = UserStatus::where('id', $this->user->estadoId);
35         return $estado->nombre;
36     }
37
38     public function setEstado($estado) {
39         $estadoId = UserStatus::where('nombre', $estado);
40         if($estadoId){
41             $this->user->estadoId = $estadoId->id;
42         }
43     }
44
45     public function getCorreo() {
46         return $this->user->correo;
47     }
48
49     public function getCarnet() {
50         return $this->student->carnet;
51     }
52
53     public function validateRegister(){
54         return $this->user->validateRegister();
55     }
56
57     public function hashPassword(){
58         return $this->user->hashPassword();
59     }
60
61     public function save(){
62         $result = $this->user->save();
63         $this->student->usuarioId = $result['id'];
64         $this->student->save();
65     }
66 }

```

Figura 4. Código de StudentUserDecorator

```

1  interface UserInterface {
2      public function getUserId();
3      public function getNombreUsuario();
4      public function getContrasenna();
5      public function getRol();
6      public function getEstado();
7      public function getCorreo();
8      public function getCarnet();
9      public function setContrasenna($contrasenna);
10     public function setEstado($estado);
11 }

```

Figura 5. Código de UserInterface

```

1  class User extends ActiveRecord implements UserInterface
2  {
3      protected static $table = 'usuario';
4      protected static $columnsDB = ['id', 'nombre', 'apellidos', 'correo', 'contrasenna', 'celular', 'campusId', 'roleId', 'estadoId', 'token', 'fechaNotificacion'];
5
6      public $id;
7      public $nombre;
8      public $apellidos;
9      public $correo;
10     public $contrasenna;
11     public $celular;
12     public $campusId;
13     public $roleId;
14     public $estadoId;
15     public $token;
16     public $fechaNotificacion;
17 }

```

Figura 6. Código de User

```

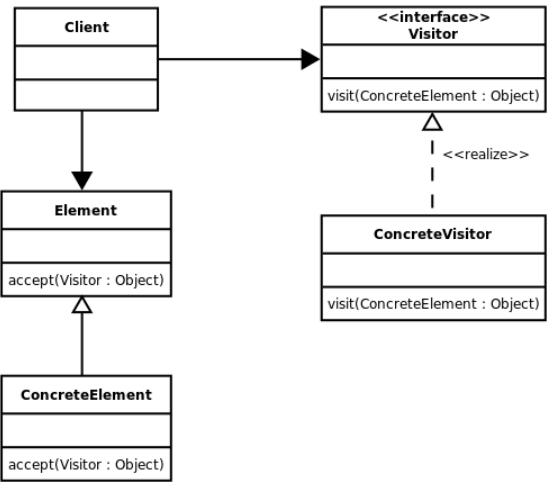
1  public static function register(Router $router){
2      $alerts = [];
3
4      if($_SERVER['REQUEST_METHOD'] === 'POST'){
5          $file = $_FILES['file']['tmp_name'];
6          $extension = pathinfo($_FILES['file']['name'], PATHINFO_EXTENSION);
7
8          $manager = new ExcelManager($file, $extension);
9          $records = $manager->getRecords();
10
11         if(!$records){
12             $alerts['error'][] = 'Archivo no válido';
13         } else {
14             foreach($records as $record){
15                 // LowerCase all Keys
16                 $record = array_change_key_case($record, CASE_LOWER);
17
18                 // Get the campusId
19                 $campus = Campus::where('nombre', $record['campus']);
20                 $record['campusId'] = $campus->id;
21
22                 // Get the roleId
23                 $rol = Role::where('nombre', 'Estudiante');
24                 $record['roleId'] = $rol->id;
25
26                 // Get the estadoId
27                 $estado = UserStatus::where('nombre', 'Activo');
28                 $record['estadoId'] = $estado->id;
29
30                 // Create the StudentUserDecorator
31                 $studentDecorator = new StudentUserDecorator($record);
32                 // debug($studentDecorator);
33
34                 // Create the user
35                 // $user = new User($record);
36                 // $user->id = null;
37                 $alerts = $studentDecorator->validateRegister();
38
39                 if(empty($alerts)){
40                     $studentDecorator->hashPassword();
41                     // debug($studentDecorator);
42                     $studentDecorator->save();
43
44                     // $result = $user->save();
45
46                     // Create the student
47                     // $student = new Student([
48                     //     'usuarioId' => $result['id'],
49                     //     'carnet' => $record['carnet']
50                     // ]);
51                     // $student->save();
52                 }
53             }
54         }
55         $alerts['success'][] = 'Estudiantes registrados correctamente';
56     }
57 }
58 $router->render('students/register', [
59     'alerts' => $alerts
60 ]);
61 }
62

```

Figura 7. Código de Register

Elementos básicos y fundamentales de la arquitectura para los patrones Visitor y Observer implementados

Patrón Visitor

<p>Diagrama Genérico</p>	 <pre> classDiagram class Client class Element { accept(Visitor : Object) } class ConcreteElement { accept(Visitor : Object) } class Visitor { <<interface>> visit(ConcreteElement : Object) } class ConcreteVisitor { visit(ConcreteElement : Object) } Client --> Visitor Element < -- ConcreteElement Visitor < .. ConcreteVisitor : <<realize>> </pre> <p>Figura 8: Diagrama genérico de patrón Visitor</p>
<p>Elementos de Patrón Visitor</p>	<p>Un patrón visitor requiere de ciertas características básicas que la componen, entre estas están el Visitor, que define una interfaz con métodos de visita específicos para cada tipo concreto de elementos, Concrete visitor, Implementa la interfaz Visitor. Contiene la lógica específica de la operación que debe realizarse cuando se visita un elemento concreto, el Element, Define una interfaz con un método Accept(visitor). Este método es crucial porque permite que un visitante interactúe con el elemento y Concrete element, que implementa la interfaz element. Contiene atributos y métodos específicos del elemento concreto, incluyendo el método accept que permite que el visitante realice operaciones sobre el elemento.</p>
<p>Elementos de la Arquitectura (Clases utilizadas)</p>	<p>Cliente:</p> <p>El Cliente interactúa con las actividades y los visitantes. Es la parte del sistema que inicia la operación del visitante sobre los elementos.</p>

	<p>Element (Interfaz):</p> <p>Esta interfaz define el método <i>accept(visitor: Visitor): void</i>, que permite que un objeto visitante realice operaciones sobre el elemento que implementa esta interfaz.</p> <p>Activity (Elemento Concreto):</p> <p>La clase <i>Activity</i> implementa la interfaz <i>Element</i>. Contiene atributos específicos de una actividad (como id, nombre, fecha, etc.) y métodos como <i>validateUpdate</i>, <i>setEstado</i>, y <i>accept</i> que permite que un Visitor interactúe con la actividad.</p> <p>Plan:</p> <p>Clase que contiene información sobre un plan y un método <i>validate</i> para verificar su estado. Se relaciona con la Activity pero no directamente con el patrón Visitor en este diagrama.</p> <p>Visitor (Interfaz):</p> <p>Define el método <i>visitActivity(activity: Activity): void</i>. Este método declara una operación que un visitante puede realizar en un objeto Activity.</p> <p>ReminderVisitor (Visitante Concreto):</p> <p>Implementa la interfaz <i>Visitor</i>. Contiene atributos como <i>currentDate</i> y <i>notificationCenter</i>, y define el método <i>visitActivity(activity: Activity): void</i>, que realiza operaciones específicas sobre una actividad, como enviar recordatorios a través del NotificationCenter.</p> <p>PublishVisitor (Visitante Concreto):</p> <p>Implementa la interfaz <i>Visitor</i>. Es otro visitante concreto que contiene atributos como <i>CurrentDate</i>. También realiza operaciones sobre una actividad, en este caso publica las actividades.</p> <p><u>Elementos utilizados por el visitor pero no</u></p>
--	--

	<p><u>son inherentemente parte del patrón:</u></p> <p>NotificationCenter:</p> <p>No es parte del patrón Visitor pero en este caso es utilizado en la arquitectura. Gestiona la lista de observadores y proporciona métodos para adjuntar, separar y notificar a los observadores sobre cambios en las actividades. Parte del Observer</p> <p>Observer (Interfaz):</p> <p>Define el método <i>update()</i>: <i>void</i>, que será llamado por el <i>NotificationCenter</i> para notificar a los observadores sobre un cambio.</p>
Conclusión de la Implementación	<p>El patrón Visitor y su implementación facilita la adición de nuevas operaciones a las clases <i>Activity</i> sin modificar su estructura, en este caso con los siguientes visitantes concretos. La interfaz <i>Visitor</i> declara el método <i>visitActivity</i>, que es implementado por la clase concreta <i>ReminderVisitor</i>. Esto permite que <i>ReminderVisitor</i> ejecute operaciones específicas, como enviar recordatorios, al visitar una instancia de <i>Activity</i>. Otra operación de la clase <i>Activity</i> es el de <i>PublishVisitor</i> que también implementa sus operaciones específicas como subir las actividades. Se implementa el patrón Visitor a la arquitectura.</p>

Patrón Observer

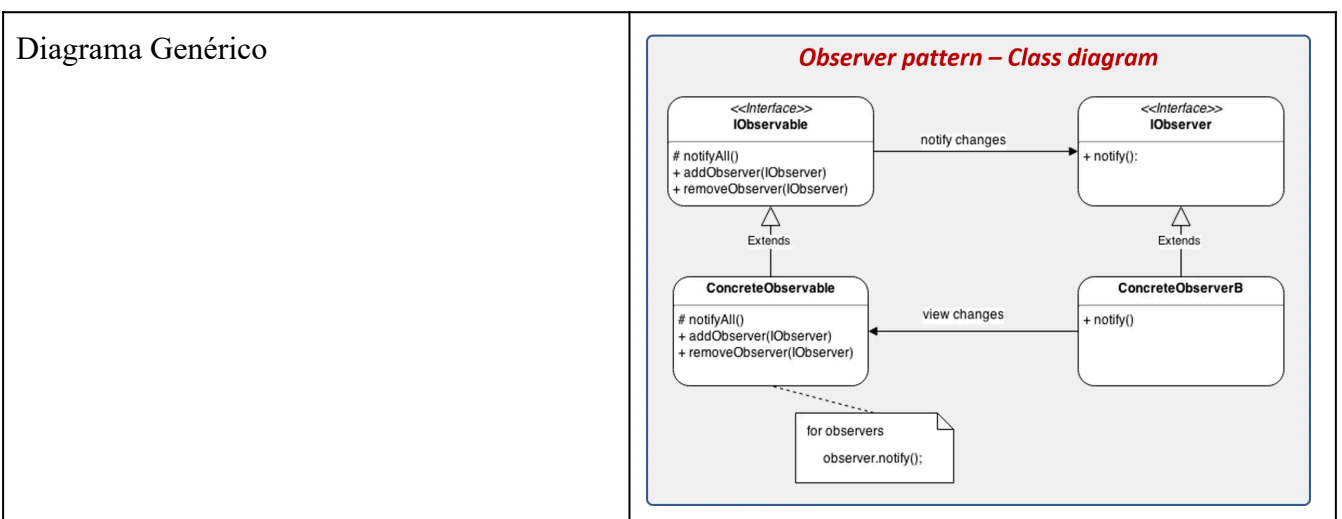


	Figura 9: Patrón genérico del patrón Observer
Elementos de Patrón Observer	<p>El observer contiene un sujeto (en este caso el observable) que contiene una lista de los observadores, el sujeto concreto, que implementa la interfaz de <i>sujeto</i>, el <i>observer</i> y mantiene el estado del cual los sujetos dependen conteniendo un método para notificarlos, está el observer que define una interfaz para los objetos que deben ser notificados sobre cambios en el sujeto y están los concrete observers estos implementan la interfaz <i>Observer</i>. Mantiene una referencia al <i>ConcreteSubject</i>, implementa el método <i>update()</i> para realizar las acciones necesarias cuando es notificado sobre cambios en el sujeto.</p>
Elementos de la Arquitectura (Clases utilizadas)	<p>Cliente:</p> <p>Interactúa con <i>Activity</i>, <i>NotificationCenter</i> y <i>StudentObserver</i>.</p> <p>Subject (Interfaz):</p> <p>El <i>Subject</i> actúa como una interfaz o clase abstracta que define los métodos necesarios para manejar a los observadores.</p> <p>NotificationCenter (Sujeto Concreto):</p> <p>Mantiene una lista de observadores. Proporciona métodos para adjuntar (<i>attach</i>), separar (<i>detach</i>), y notificar (<i>notify</i>) a los observadores sobre cambios. Contiene métodos adicionales como <i>createMessage</i> y <i>publish</i>.</p> <p>Observer (Interfaz):</p> <p>Define el método <i>update()</i>, que es llamado por el <i>NotificationCenter</i> para notificar a los observadores sobre un cambio.</p> <p>StudentObserver (Observador Concreto):</p> <p>Implementa la interfaz <i>Observer</i>. Contiene la lógica específica que se ejecuta cuando es notificado sobre un cambio en una actividad a través del <i>NotificationCenter</i>.</p>

	<p>Activity:</p> <p>Contiene información sobre una actividad, como id, nombre, fecha, etc. Incluye métodos como <i>validateUpdate</i>, <i>setEstado</i>, y <i>accept</i> para interactuar con <i>Visitor</i>. Se relaciona con <i>NotificationCenter</i> para notificar cambios.</p> <p>Plan:</p> <p>Contiene información sobre un plan y un método <i>validate</i> para verificar su estado. Relacionado con <i>Activity</i> pero no directamente con el patrón Observer en este diagrama.</p>
Conclusión de la Implementación	<p>El patrón Observer se utiliza para gestionar la notificación de cambios en las actividades (Activity) a los observadores registrados. El <i>NotificationCenter</i> actúa como el sujeto, manteniendo una lista de observadores que implementan la interfaz Observer. Cuando una actividad cambia, el <i>NotificationCenter</i> llama a su método <i>notify</i>, que invoca el método <i>update</i> de cada observador registrado. En este caso, el <i>StudentObserver</i>, que es un observador concreto, implementa el método <i>update</i> para realizar acciones específicas cuando es notificado sobre un cambio. Esto permite que los Activity se mantengan independientes y desacoplados de los observadores que reaccionan a sus cambios, implementando el patrón observer en nuestra arquitectura.</p>

Nota: NotificationCenter y Observer son compartidos por ambos patrones de comportamiento pero solo es parte del patrón observer. En el visitor es solo parte de su funcionamiento pero no del patrón en sí.

Diagrama de clases (UML) actualizado

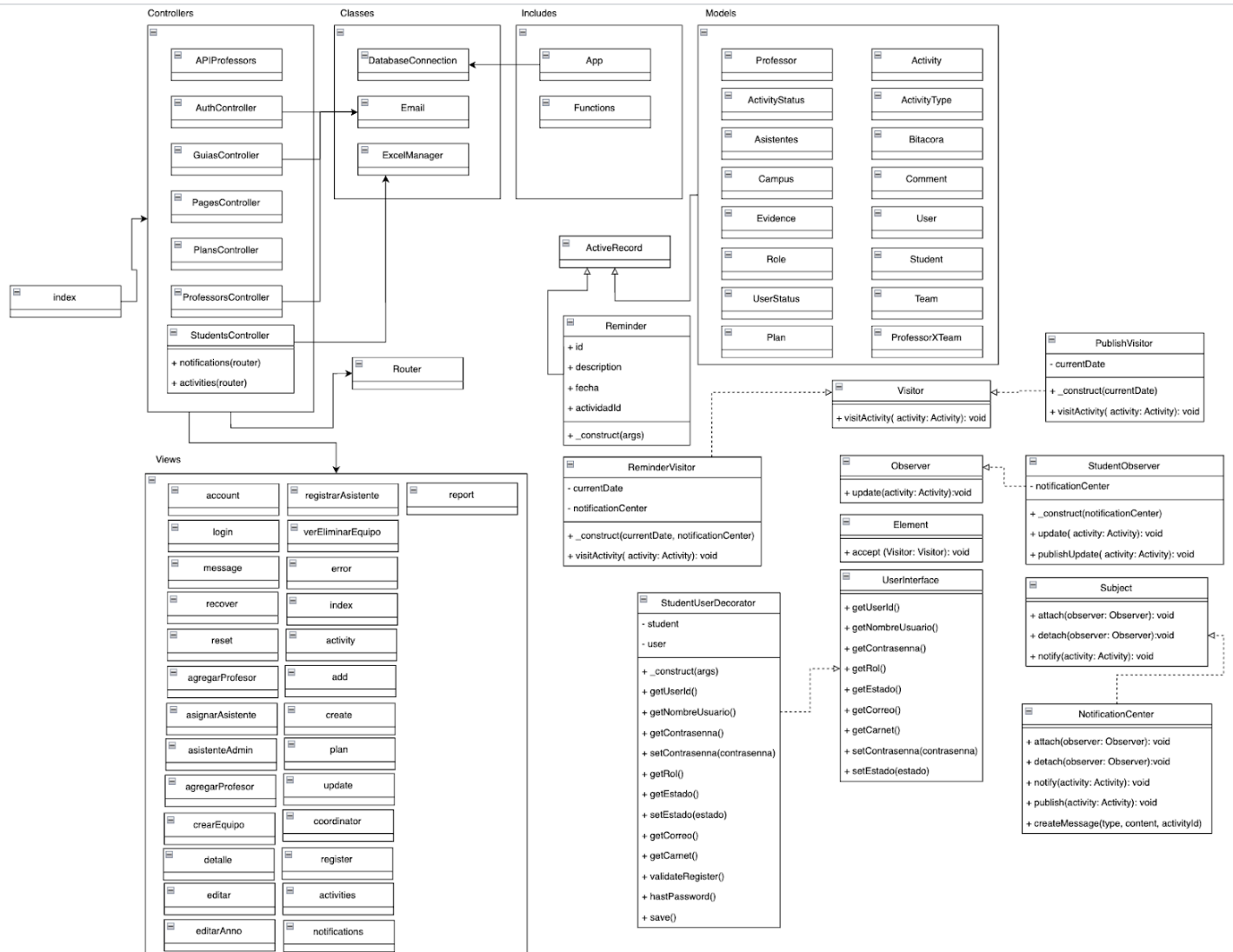


Figura 10: Diagrama de Clases actualizado (Con los patrones agregados)

Cuadro de análisis de resultados de funcionalidades solicitadas

Funcionalidad	Calificación (0-100)
Plan de trabajo por equipo guía de profesores	100
Información de estudiantes	100
Estudiantes como nuevos usuarios	100
Servicio de notificaciones asociado al perfil del estudiante	100
activaciones y recordatorios de actividades del plan	100
Notificaciones de las actividades a estudiantes	100
El “Visitante”	100
El “Observador”	100
Estados de actividades	100
Perfil del Estudiante	100

Cuadro de lecciones aprendidas individualmente

Miembro	Lecciones Aprendidas
Erika Michelle Cerdas Mejías	<p>Durante el desarrollo de este proyecto de creación de una página web para la gestión de grupos guías y estudiantes, aprendí la importancia de combinar conocimientos teóricos con habilidades prácticas y trabajar en equipo de manera efectiva. Utilizando herramientas de diseño de software, como diagramas UML, y tecnologías de desarrollo web, en particular PHP, demostró ser esencial para la implementación exitosa del proyecto. La aplicación de patrones de diseño estructurales y de comportamiento, como Visitor y Observer, no solo mejoró la flexibilidad y mantenibilidad del software, sino que también permitió abordar problemas complejos de manera eficiente. Implementar estos patrones en PHP facilitó la gestión jerárquica de los grupos y las actualizaciones en</p>

	<p>tiempo real de los datos.</p> <p>Además, la experiencia de trabajar en equipo destacó la relevancia de una comunicación clara y constante. Esta experiencia reafirmó la importancia de combinar teoría y práctica, así como la necesidad de habilidades interpersonales y de gestión en el desarrollo de software.</p>
Leonardo Céspedes Tenorio	<p>La realización de este proyecto me permitió perfeccionar algunas habilidades relacionadas con el desarrollo de aplicaciones web, específicamente en el web stack que utilizamos. Así mismo, esta última fase me permitió comprender de mejor manera cómo funcionan algunos de los patrones vistos en clase de mejor manera, pues fue necesario buscar la manera más ideal de implementarlos en nuestra aplicación. El proceso de buscar cuál patrón estructural se adapta mejor a la aplicación también fue bastante enriquecedor, pues fue necesario analizar a detalle tanto cada patrón como la arquitectura que ya estaba presente para determinar cuál tenía el mejor equilibrio entre facilidad de implementación y utilidad dentro de la aplicación. Finalmente, en el proceso de adaptación de la fase 2 para implementar lo solicitado en esta última fase se dejó en claro las habilidades de resolución de conflictos del equipo.</p> <p>Otras habilidades más enfocadas en el desarrollo de software en general también fueron mejoradas o recordadas como el diseño de diagramas UML. Así mismo, las habilidades blandas y de comunicación jugaron un papel fundamental dentro del desarrollo del proyecto, pues al tratarse de un trabajo grupal, este tipo de habilidades permiten al equipo acoplarse de la mejor manera para llevar a cabo las tareas eficientemente.</p>
Kevin Chang Chang	<p>En la elaboración de nuestro proyecto creando un sitio web para el manejo de actividades y planes con grupos en el TEC aprendí diferentes habilidades y consideraciones que hay que tener para poder llevar a cabo este tipo de trabajo. En cuanto a tecnologías, utilizamos herramientas similares a otros semestres así que en ese aspecto no hubo mucho cambio, lo que sí mejoró fue mi gestión de diferentes problemas con diferentes enfoques. Más con la introducción de patrones de diseño, tuve que analizar el problemas y las posibles soluciones de los problemas dados</p>

	<p>tomando en cuenta estos nuevos conocimientos del curso. Mi aprendizaje ocurrió en la necesidad de ver el proyecto con una perspectiva diferente para poder implementar los diferentes patrones de diseño entendiendo su función, su importancia en el contexto y su implementación. Esto me dio un punto de vista distinto en comparación a otros cursos que he tenido en cuanto al desarrollo de un producto de software.</p> <p>Otro aspecto importante que aprendí fue la organización del trabajo especialmente en equipo. En otros cursos ya hemos trabajado con estos aspectos pero en cada proyecto que hacemos tenemos un enfoque diferente de trabajo y por lo tanto una gestión distinta del equipo. En este caso fué importante la comunicación en términos de los roles de cada miembro y coordinar cómo íbamos a juntar nuestro trabajo en un solo proyecto coherente.</p> <p>Finalmente aprendí la importancia de tener bien claro todas las planificaciones del proyecto en cuanto a la organización de diagramas y prototipos esto disminuye el trabajo necesario cuando se implementa el proyecto final.</p>
Frankmin Feng Zhong	<p>Esta tercera fase terminó siendo más retadora de lo que uno esperaría. Usualmente, lo que la mayoría de gente acostumbra a hacer es aplicar el cambio directamente en sobre la arquitectura, sin importar si terminan siendo cambios radicales en la estructura de los objetos. Esto puede ser muy peligroso, debido a que cada cambio podrá requerir de modificaciones en el código existente, lo que aumenta el riesgo de introducir nuevos errores. En este escenario, el patrón visitor resultó ser de mucha utilidad, ya que nos permite agregar nuevas operaciones a estructuras de objetos sin modificar sus clases. De esta manera, se puede respetar los principios SOLID.</p> <p>Para asegurar que los patrones que se van a aplicar se usen de manera efectiva, hay que ser críticos y crear con antelación los diagramas de UML para notar bien los cambios a realizar y la manera en la que se va a afectar toda la aplicación.</p> <p>En cuanto las lecciones aprendidas de los manejo de las tecnologías utilizadas, es recomendable trabajar con tecnologías con las que se haya tenido experiencia previamente, puesto que no solo hará que el proceso de</p>

	<p>construcción sea mucho más fluido y eficiente, sino también, permite escalar la aplicación de manera mucho más fácil y segura. Un buen manejo de las tecnologías permite identificar los puntos críticos y aplicar los patrones adecuados para escalar la aplicación.</p> <p>Otro aspecto importante aprendido, es la importancia de tener buena comunicación a la hora de trabajar en equipos. Es de suma importancia que todos los integrantes estén enterados y actualizados de lo que se desea hacer y cómo es que se va a realizar. Todas las personas trabajamos de manera distinta, sin embargo, el hecho de usar diagramas y tener buena documentación, puede ayudar a reducir la ambigüedad y la diferencia de expectativas dentro de un grupo.</p>
--	---

Repositorio de Git

HTTPS: <https://github.com/LeonardoC1302/IniciaTEC.git>

SSH: [git@github.com](https://github.com/LeonardoC1302/IniciaTEC.git):LeonardoC1302/IniciaTEC.git

CLI: gh repo clone LeonardoC1302/IniciaTEC